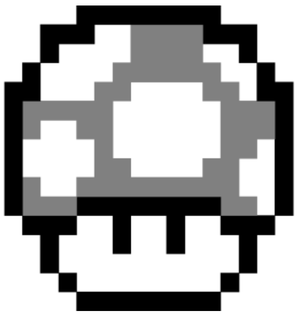
CS1010S Programming Methodology

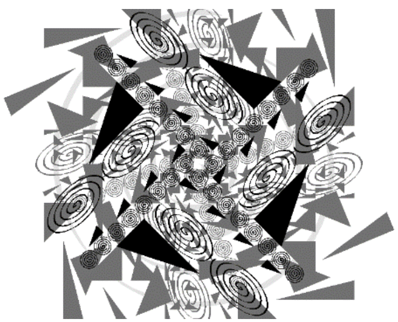
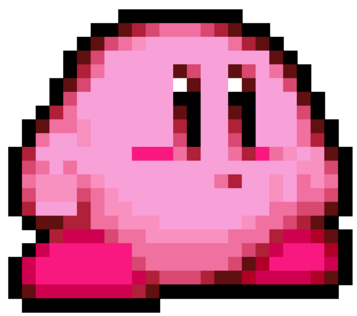
Lecture 6
Working with Sequences

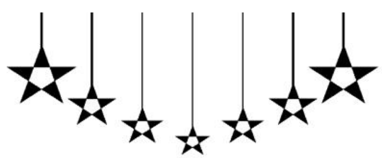
18 Feb 2014

2D & 3D Rune Contest

2D Runes - Notable Mention






- Huang Weiqi Victor
- Guan Xibeijia
- Adrian Tan Hong Ji
- Zhang Peixu

+100 XP

2D Runes – Third Place

+200 XP

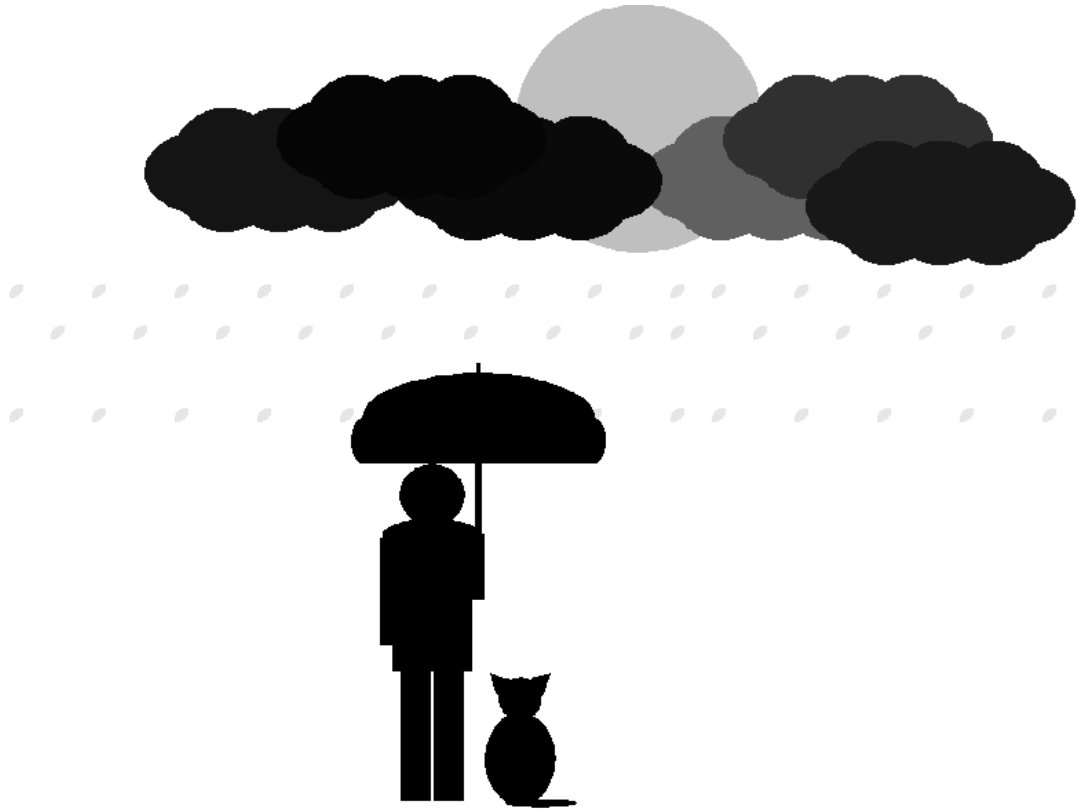
Wang Si Qi

2D Runes – Second Place

+300 XP

Zhang Peixu

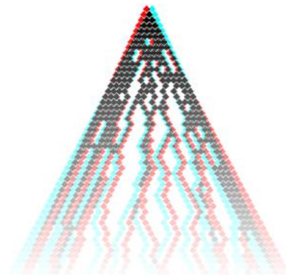
2D Runes – First Place

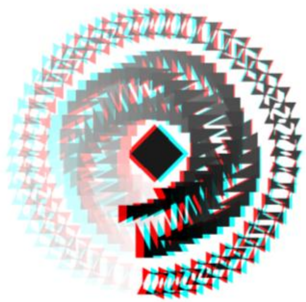


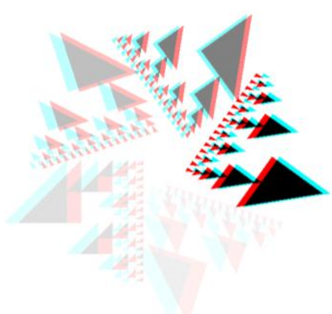
+400 XP

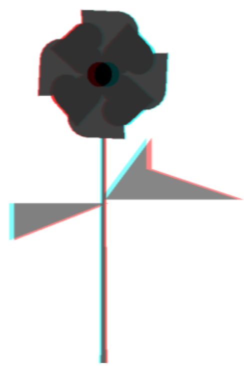
Guan Xibeijia

3D Anaglyphs – Notable Mentions









- Adrian Tan Hong Ji
- Wang Si Qi
- Tan Wei Liang
- Benjamin Ong
- Yeo Xin Yi
- Muhammad Haikal

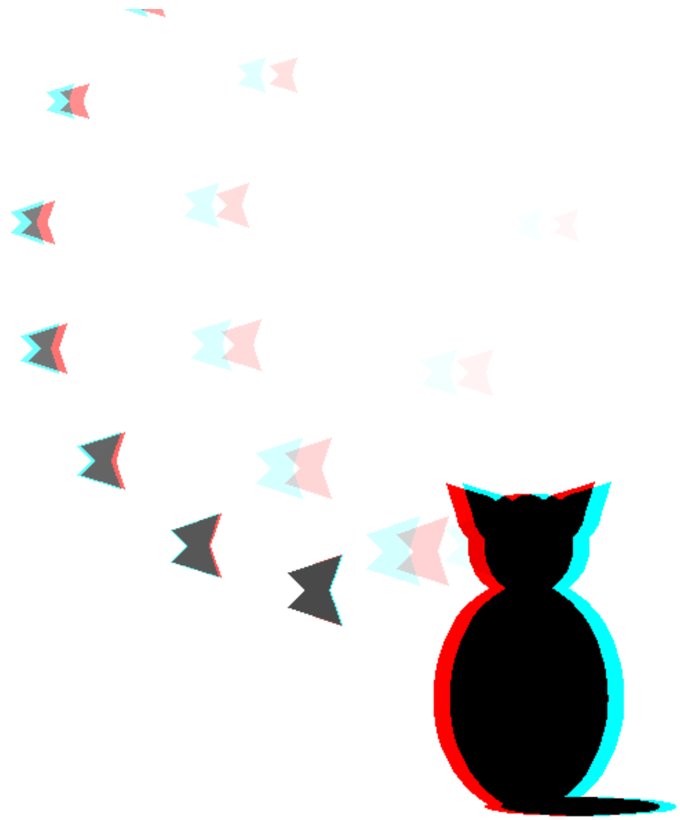
+100 XP

3D Anaglyphs – Third Place

+200 XP

Li Jiaxin Cassandra

3D Anaglyphs – Second Place



+300
XP

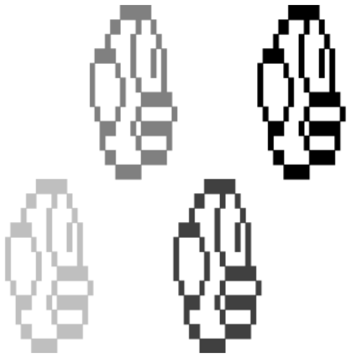
Guan Xibeijia

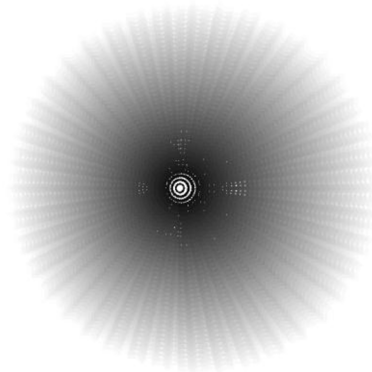
3D Anaglyphs – First Place

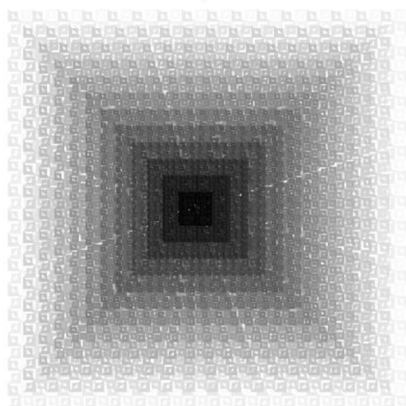
+400 XP

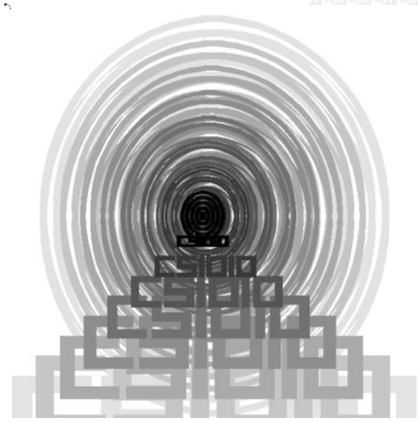
Jacinda Siew Xinying

3D Hollusion – Notable Mentions





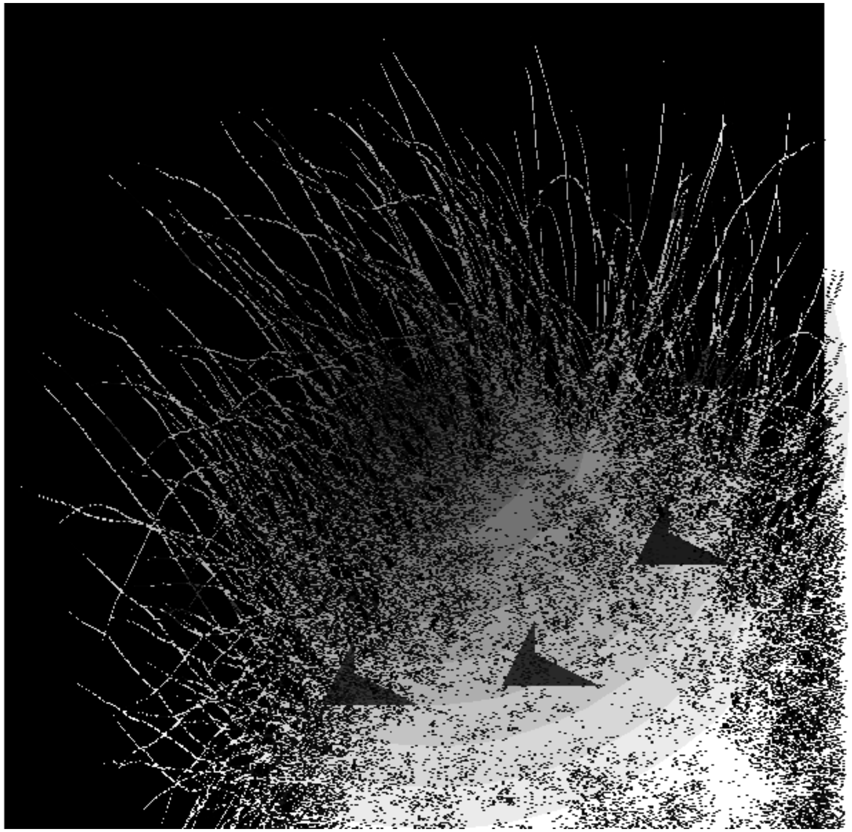




- Fang Gian Yao
- Samuel Tan
- Lee Jia Hui
- Liang Tian Ze
- Lester Sim

+100 XP

3D Hollusion – Third Place



+200 XP

Franklin Leonardo

3D Hollusion – Second Place

+300 XP

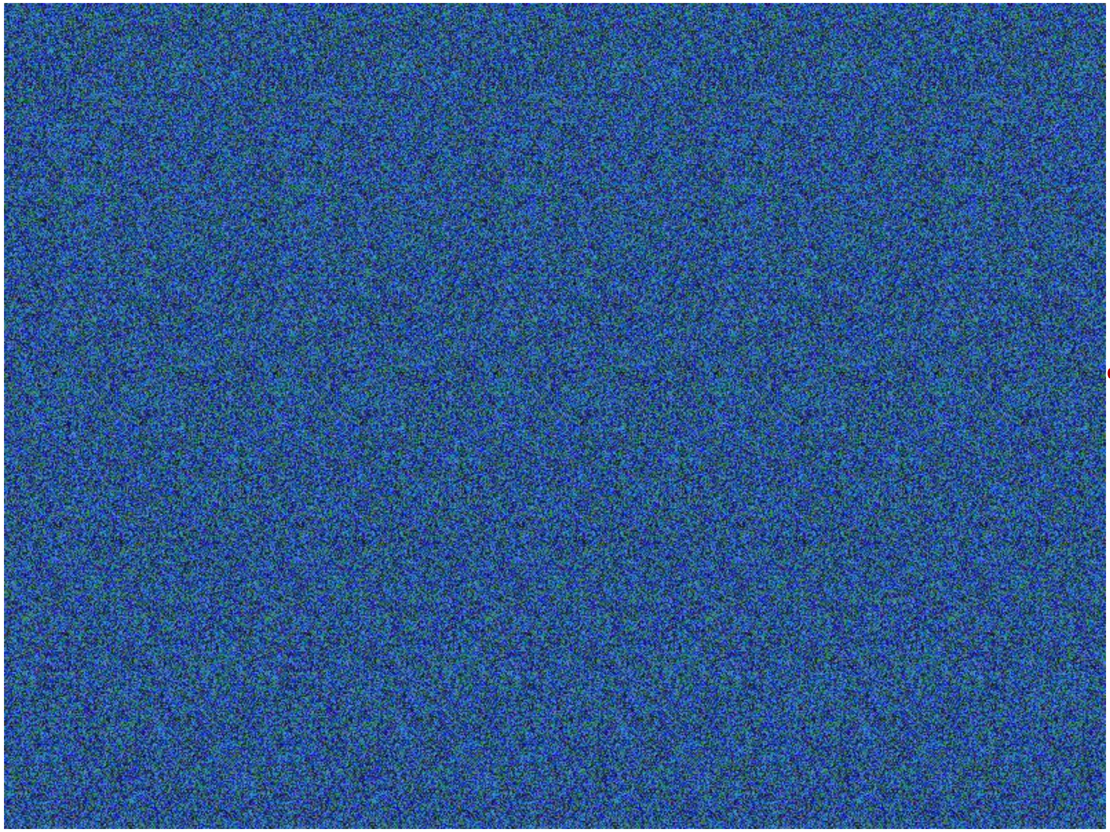
Zhang Xiaoyue

3D Hollusion – First Place

+400 XP

Yu Yingru

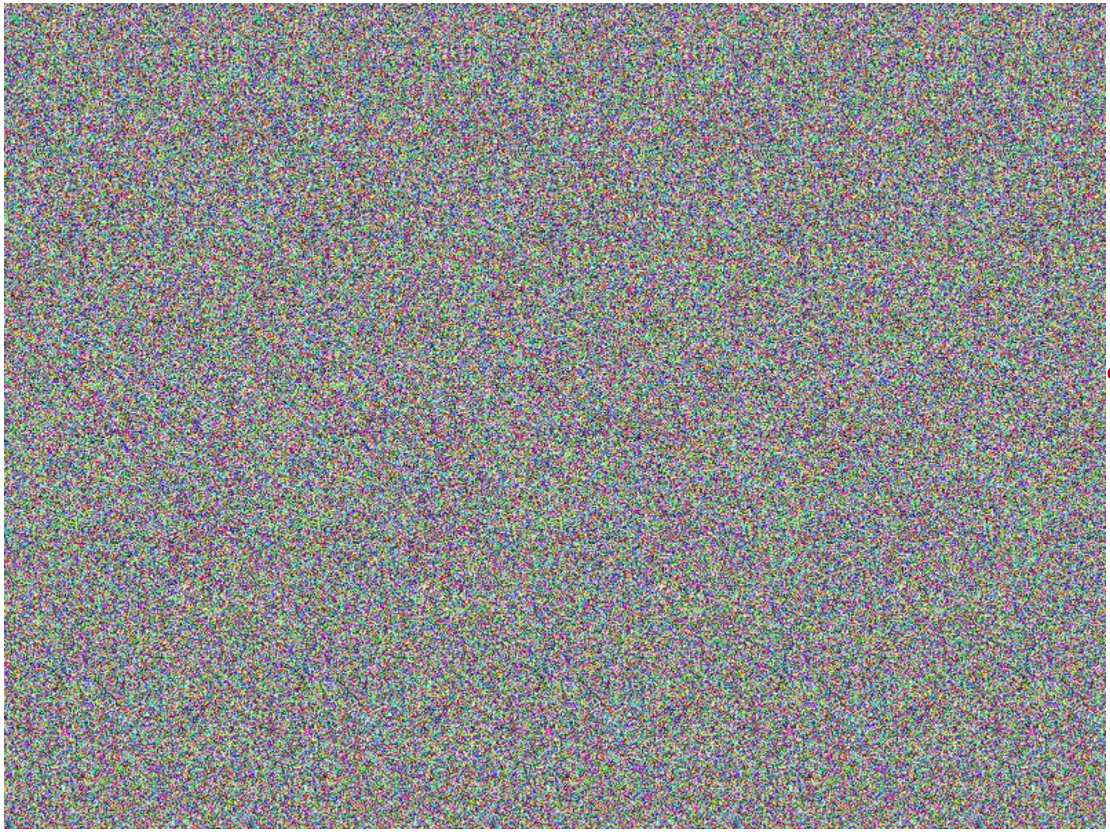
3D Stereogram - Notable Mention



+100 XP

Zhang Peixu

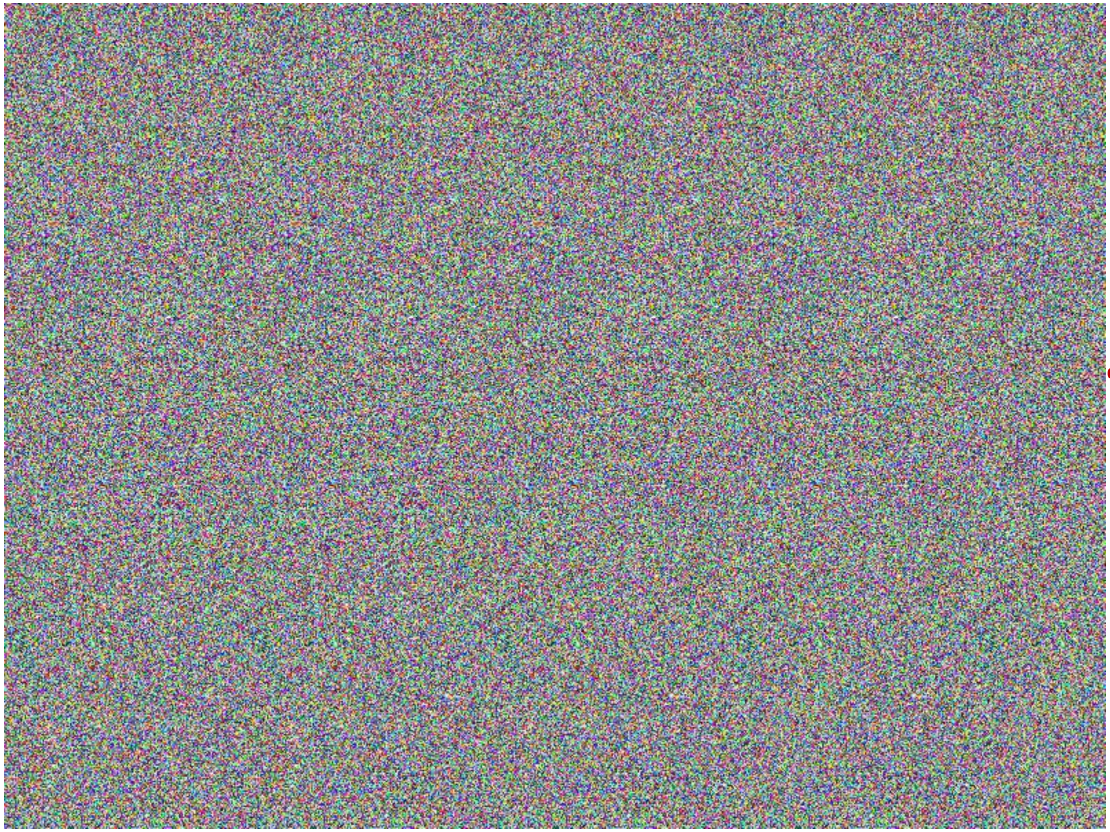
3D Stereogram – Second Place



+300 XP

Darren Wee Thai Yuan

3D Stereogram – First Place



+400 XP

Chan Jia Hui Isabella

Make-up Recitation

- Monday 23 Feb (COM2-04-02)
  - 11 – 12 noon
  - 1 – 2 pm
- Tuesday 24 Feb (COM1-02-01)
  - 10 – 11 am
  - 11 – 12 noon
  - 1 – 2 pm
- Wednesday 25 Feb (COM1-02-04)
  - 10 – 11 am

E-Learning Week

Midterm Exam

- Venue: MPSH 1
- Open-sheet exam (no laptops!)
  - 1 x A4 sheet (both sides)
- Scope: everything up to and including Lecture 5 (Data Abstraction)
- Past Year Exams have been uploaded to Coursemology

Midterm Exam

- Python Expressions
- Solving Problems with Recursion/Iteration
  - Order of Growth
- Higher Order Functions
- Data Abstraction
  - Define new Abstract Data Type + Operations

Only 15%

Don’t Stress

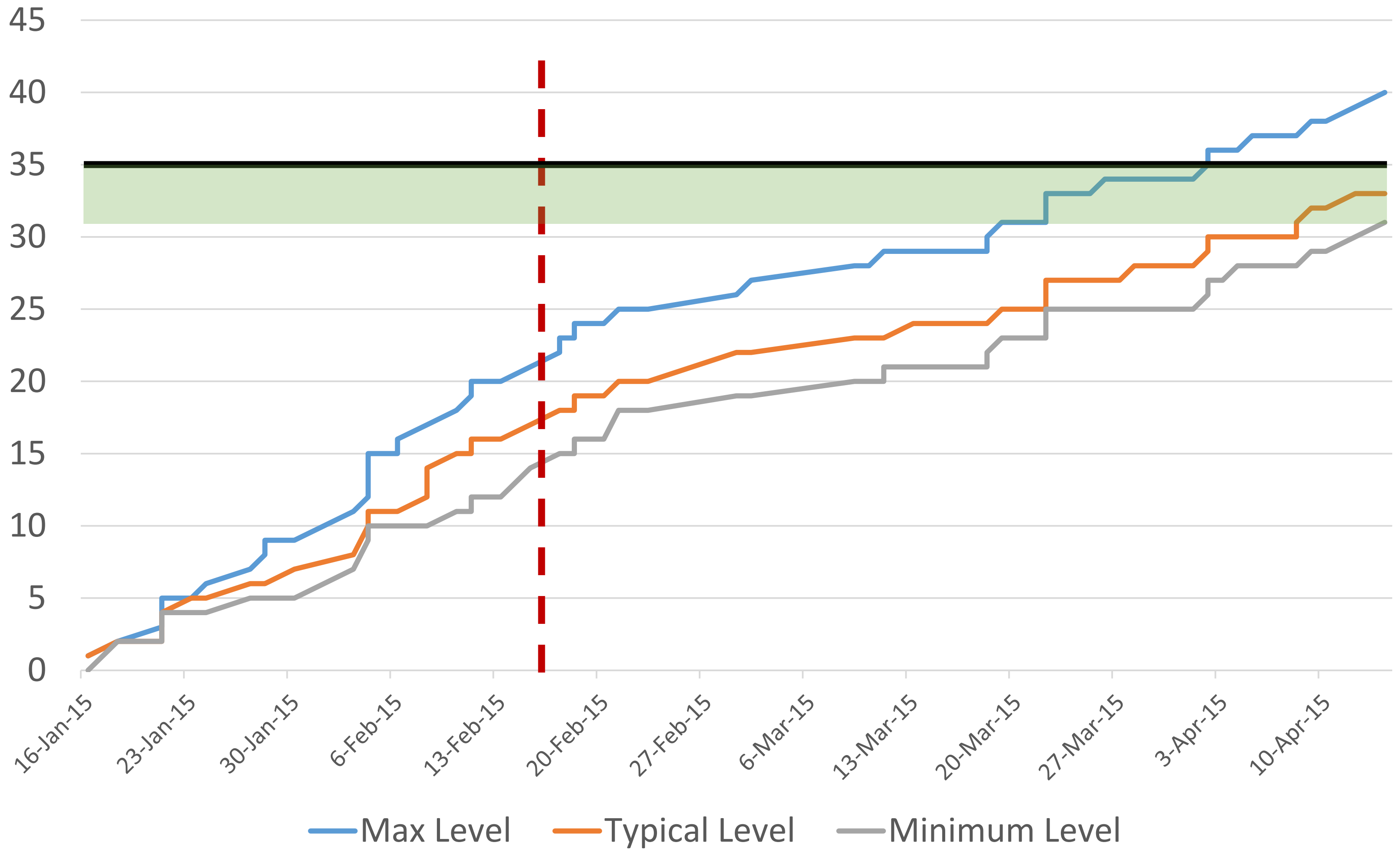
Help is Coming

- Remedial Sessions
  - 24 Feb (Tue), 6:30 – 8:30 pm
  - 27 Feb (Sat), 10:30 – 12 pm
- Exam Review
  - 26 Feb (Thurs)
- "Desperado" Session
  - 3 Mar (Tues)

No Tutorials & Recitations

- No Recitations on midterm week
- Tutors will still be at the lab on Mon/Tues during tutorial times for consultation

EXP curve



45
40
35
30
25
20
15
10
5
0
16-Jan-15
23-Jan-15
30-Jan-15
6-Feb-15
13-Feb-15
20-Feb-15
27-Feb-15
6-Mar-15
13-Mar-15
20-Mar-15
27-Mar-15
3-Apr-15
10-Apr-15
Max Level
Typical Level
Minimum Level

Today’s Agenda

- Processing Sequences
  - Recursion & Iteration
- Tree as nested sequences
  - Hierarchical structures
- Signal-processing view of Computations
- Working with Files

Recap: Data Abstraction

- Abstracts away irrelevant details, exposes what is necessary
- Separates usage from implementation.
- Captures common programming patterns
- Serves as a building block for other compound data.

Key idea

- Decide on an internal representation of the Abstract Data Type (ADT) Tuple!
- Write functions that operate on that new ADT

Key insight: nobody needs to know your internal representation to use your ADT

Guidelines for Creating Compound Data

- Constructors
  - To create compound data from primitive data
- Selector (Accessors)
  - To access individual components of compound data
- Predicates
  - To ask (true/false) questions about compound data
- Printers
  - To display compound data in human-readable form

Sequences

- Sequential data, represented by tuples
- Get the first element of the list:

```
seq[0]
```

- Get the rest of the elements:

```
seq[1:]
```

- If a list is a tuple containing a single integer 4:

```
seq = (4,)
seq[0] → 4
seq[1:]  → ()
```

Reversing a Sequence

```
def reverse(seq):
    if seq == ():
        return ()
    else:
        return reverse(seq[1:]) + (seq[0],)
```

Recursive

- Notice that `(seq[0],)` is a tuple and not an integer
- Can only concatinate tuples with tuples

Orders of Growth

```
def reverse(seq):
    result = ()
    for item in seq:
        result = (item,) + result
    return result
```

Iterative

`tuple1 + tuple2` takes `len(tuple1) + len(tuple2)` steps!

	Time	Space
• Orders of growth:		
- Recursive version:	$O(n^2)$	$O(n^2)$
- Iterative version:	$O(n^2)$	$O(n)$

Key Idea:

Handle the First Element and then the Rest

Iterate down the sequence!

Scaling a sequence

Suppose we want to scale all the elements of a sequence by some factor

`scale_seq((1, 2, 3, 4), 3)` → `(3, 6, 9, 12)`

```
def scale_seq(seq, factor):
    if seq == ():
        return ()
    else:
        return (seq[0] * factor,) +
                scale_seq(seq[1:], factor)
```

Time? $O(n^2)$

Space? $O(n^2)$

Scaling a sequence (iterative)

Suppose we want to scale all the elements of a sequence by some factor

```
scale_seq((1, 2, 3, 4), 3) → (3, 6, 9, 12)
```

```
def scale_seq(seq, factor):
    result = ()
    for element in seq:
        result = result + (element * factor,)
    return result
```

Time? $O(n^2)$

Space? $O(n)$

Squaring a sequence

Given a sequence, we want to return a sequence of the squares of all elements.

`square_seq((1, 2, 3, 4))` → `(1, 4, 9, 16)`

```
def square_seq(seq):
    if seq == ():
        return ()
    else:
        return (seq[0] ** 2, ) +
                square_seq(seq[1:])
```

Time? $O(n^2)$

Space? $O(n)$

Homework: Do this iteratively

Looking for patterns ….

```
def scale_seq(seq, factor):
    if seq == ():
        return ()
    else:
        return (seq[0] * factor,) +
                scale_seq(seq[1:], factor)

def square_seq(seq):
    if seq == ():
        return ()
    else:
        return (seq[0] ** 2,) +
                square_seq(seq[1:])
```

Higher-order function!!

Mapping

Often, we want to perform the same operation on every element of a list.

$$\begin{array}{cccccc} (& \mathtt{a} & \mathtt{b} & \mathtt{c} & \mathtt{d} &) \\ f & \downarrow & \downarrow & \downarrow & \downarrow & \\ (& f(\mathtt{a}) & f(\mathtt{b}) & f(\mathtt{c}) & f(\mathtt{d}) &) \end{array}$$

This is called *mapping*.

Mapping

```
def map(fn, seq):
    if seq == ():
        return ()
    else:
        return (fn(seq[0]), ) + map(fn, seq[1:])
```

Note: this will overwrite
the default Python map function!

Scaling a list by a factor

```
def scale_seq(seq, factor):
    return map(lambda x: x * factor, seq))
```

Examples

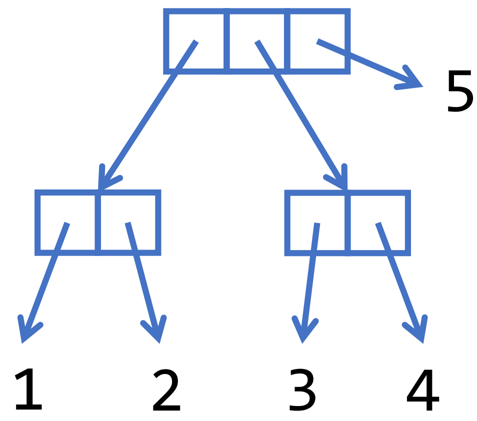
```
map(abs, (-10, 2.5, -11.6, 17))
→ (10, 2.5, 11.6, 17)

map(square, (1, 2, 3, 4))
→ (1, 4, 9, 16)

map(cube, (1, 2, 3, 4))
→ (1, 8, 27, 64)
```

Trees

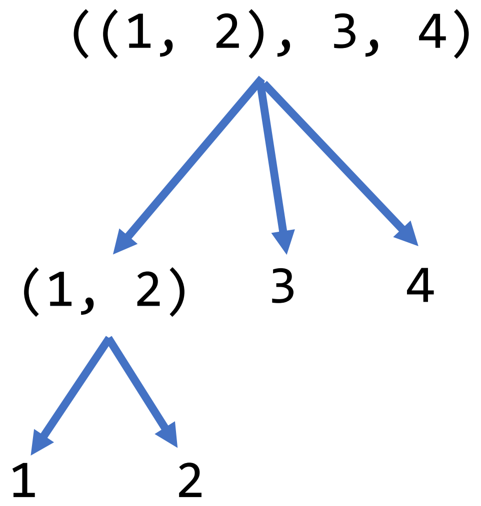
Trees are sequences of sequences and single elements

- This is possible because of the closure property: we can include a sequence as an element of another sequence
- This allows us to build hierarchical structures, e.g. trees.

```
((1, 2), (3, 4), 5)
```



Examples



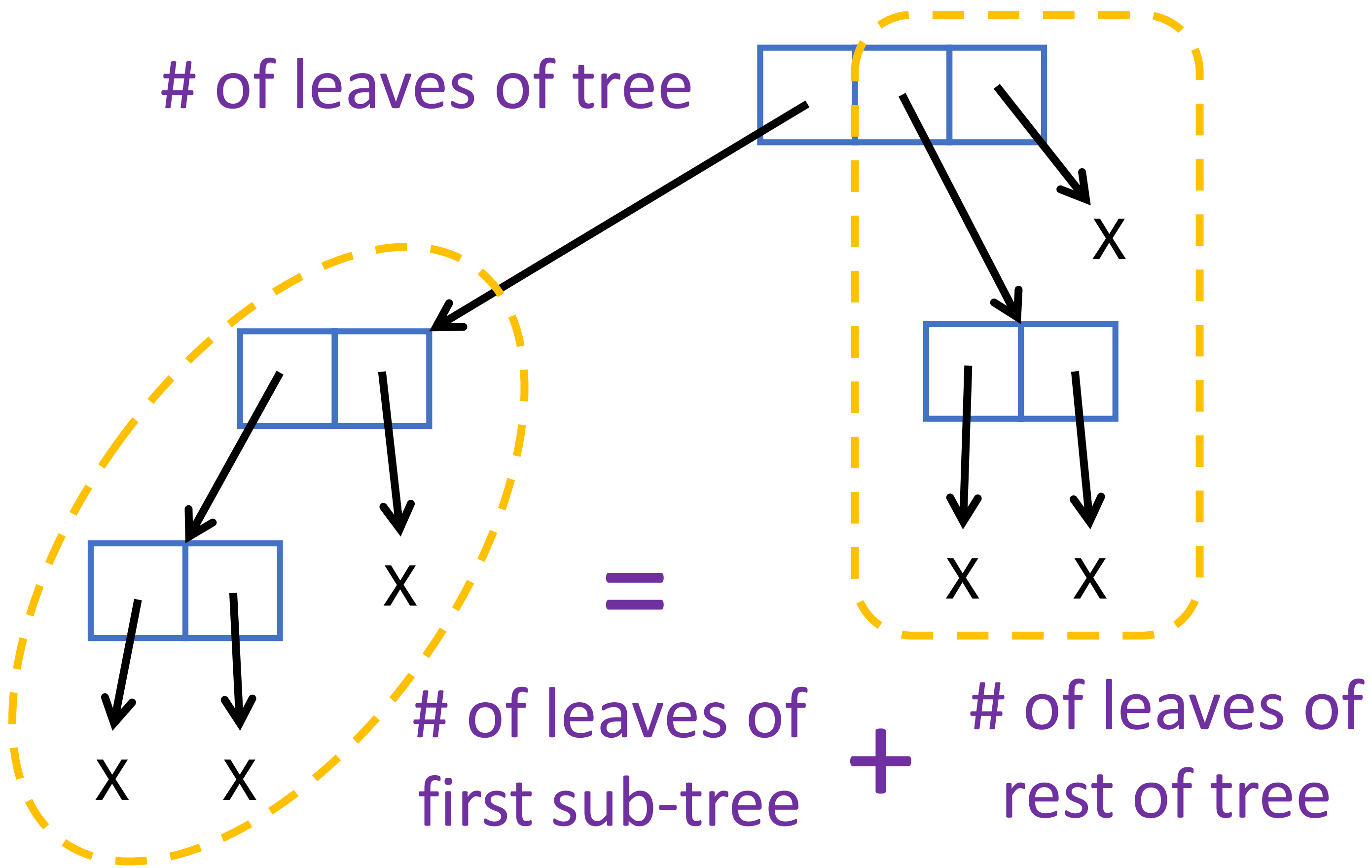

```
x = ((1, 2), 3, 4)
len(x) → 3
count_leaves(x)  → 4
(x, x)
→ (((1, 2), 3, 4), ((1, 2), 3, 4))
len((x, x))

→ 2
count_leaves((x, x))

→ 8
```

How would we count the leaves?

RECURSION!

Recurrence Relation

Observation:



Recursion

In other words,

```
count_leaves(tree) =
count_leaves(tree[0]) +
count_leaves(tree[1:])
```

Base Case:

If tree is empty

Zero!

Another Base Case

Observe:

Possible for the head or tail to be a leaf!

$$\text{Leaf} \Rightarrow +1$$

Summary

Strategy:

- If tree is empty, then 0
- Another base case:
  - tree is a leaf, then count as 1
- Count this, and add to:
  - `tail` also a tree, so recursively count this

Count Leaves

```
def count_leaves(tree):
    if tree == ():
        return 0
    elif is_leaf(tree):
        return 1
    else:
        return count_leaves(tree[0])
        + count_leaves(tree[1:])
```

What are leaves

Remember `type()` in Lecture 1:

```
>>> t = (1, 2, 3)
>>> type(t)
<class 'tuple'>

>>> type(t) == tuple
True

def is_leaf(item):
    return type(item) != tuple
```

Mapping over trees

Suppose we want to scale each leaf by a factor, i.e.

```
mytree → (1, (2, (3, 4), 5), (6, 7))

scale_tree(mytree, 10)
  → (10, (20, (30, 40), 50), (60, 70))
```

Strategy

- Since tree is a sequence of sequences, we can map over each element in a tree.
- Each element is a subtree, which we recursively scale, and return sequence of results.
- Base case: if tree is a leaf, multiply by factor

Mapping over trees

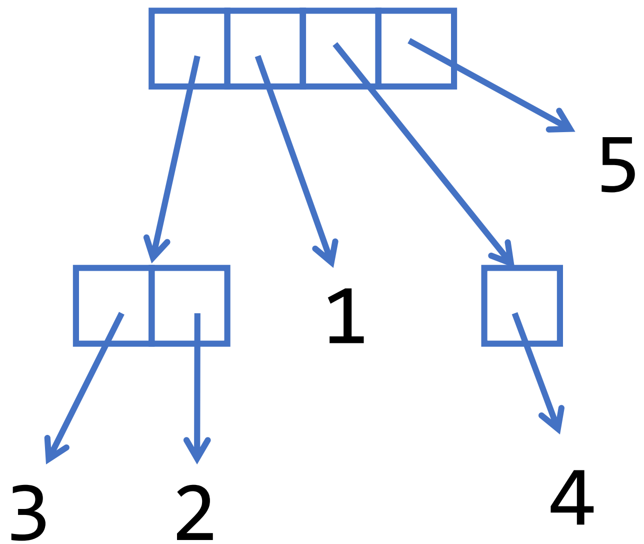
```
def scale_tree(tree, factor):
    def scale_func(subtree):
        if is_leaf(subtree):
            return factor * subtree
        else:
            return scale_tree(subtree, factor)
    return map(scale_func, tree)
```

Compare with:

```
def count_leaves(tree):
    if tree == ():
        return 0
    elif is_leaf(tree):
        return 1
    else:
        return count_leaves(tree[0]) + count_leaves(tree[1:])
```

Let’s see what scale_tree does

```
tree = ((3, 2), 1, (4,), 5)
```



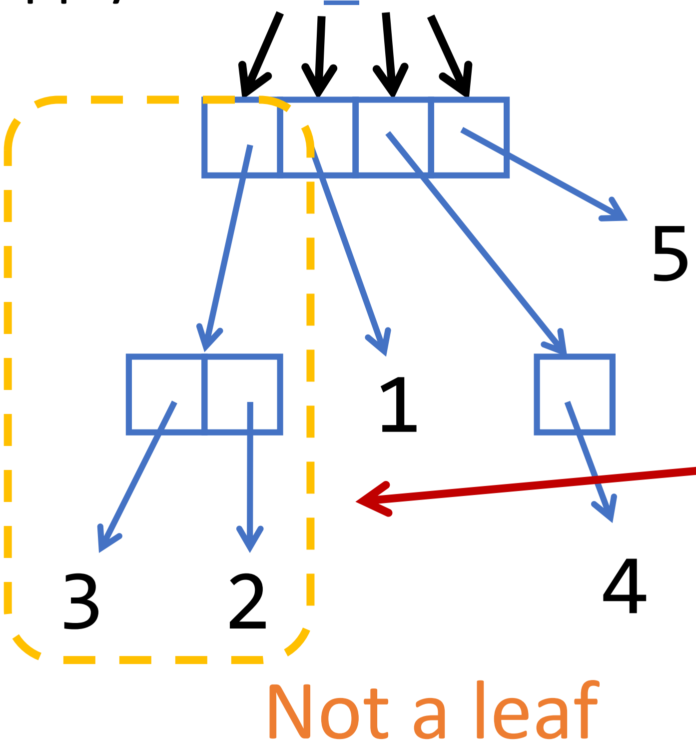

```
def scale_tree(tree, factor):
  def scale_func(subtree):
    if is_leaf(subtree):
       return factor*subtree
    else:
       return scale_tree(subtree,
                         factor)
  return map(scale_func, tree)
```

Suppose we do scale_tree(tree, 2)

Let's see what `scale_tree` does

`tree = ((3, 2), 1, (4,), 5)`

Apply `scale_func` to each element

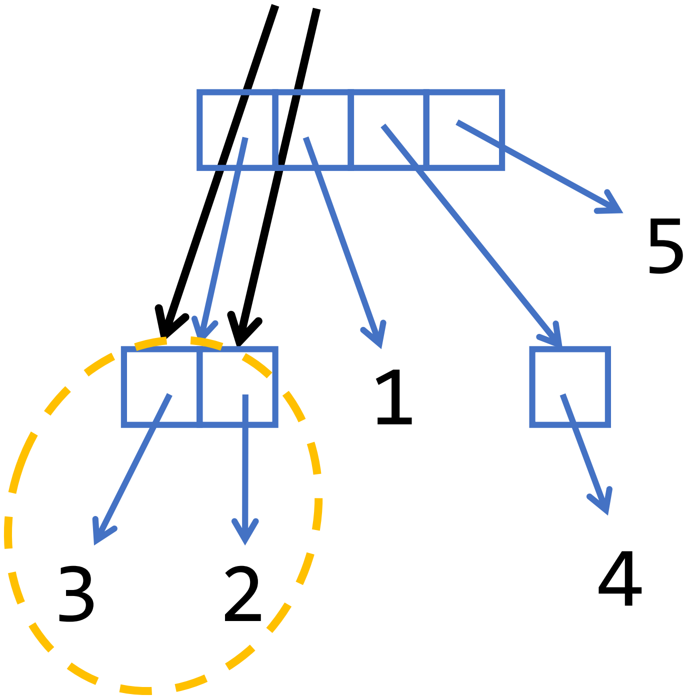



```
def scale_tree(tree, factor):
    def scale_func(subtree):
        if is_leaf(subtree):
            return factor*subtree
        else:
            return scale_tree(subtree,
                              factor)
    return map(scale_func, tree)
```

Let’s see what `scale_tree` does

`tree = ((3, 2), 1, (4,), 5)`

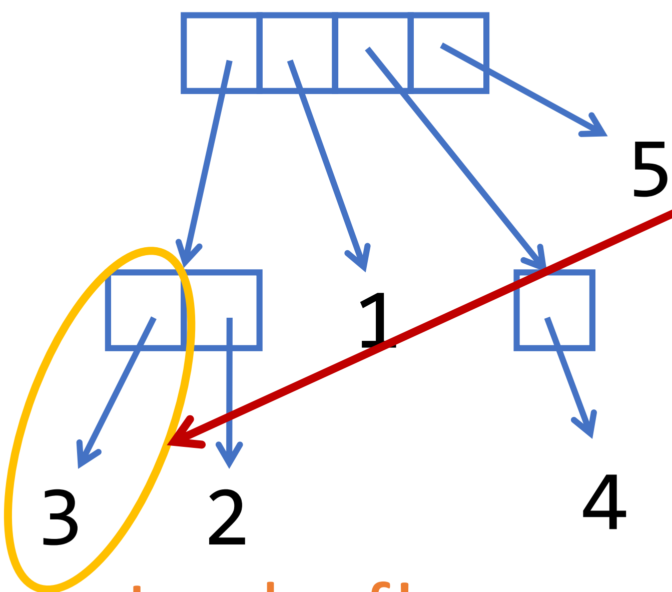
Apply `scale_func` to each element




```
def scale_tree(tree, factor):
    def scale_func(subtree):
        if is_leaf(subtree):
            return factor*subtree
        else:
            return scale_tree(subtree,
                              factor)
    return map(scale_func, tree)
```

Let’s see what scale_tree does

```
tree = ((3, 2), 1, (4,), 5)
```

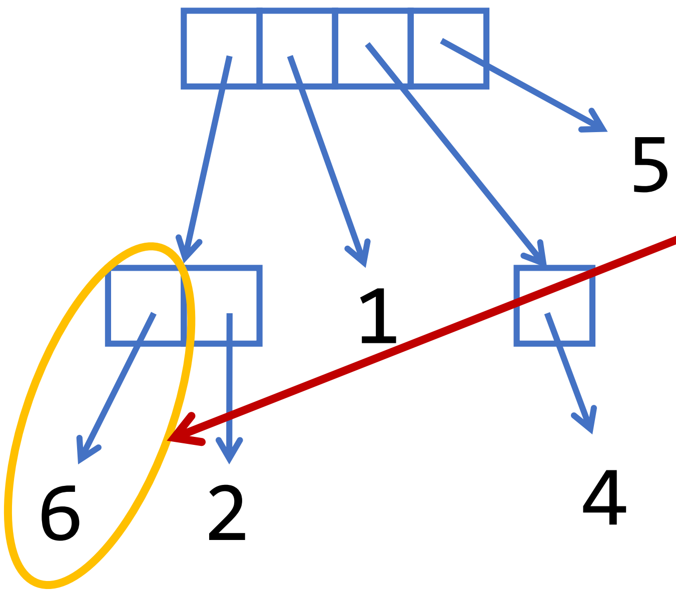


Is a leaf!

```
def scale_tree(tree, factor):
    def scale_func(subtree):
        if is_leaf(subtree):
            return factor*subtree
        else:
            return scale_tree(subtree,
                              factor)
    return map(scale_func, tree)
```

Let's see what `scale_tree` does

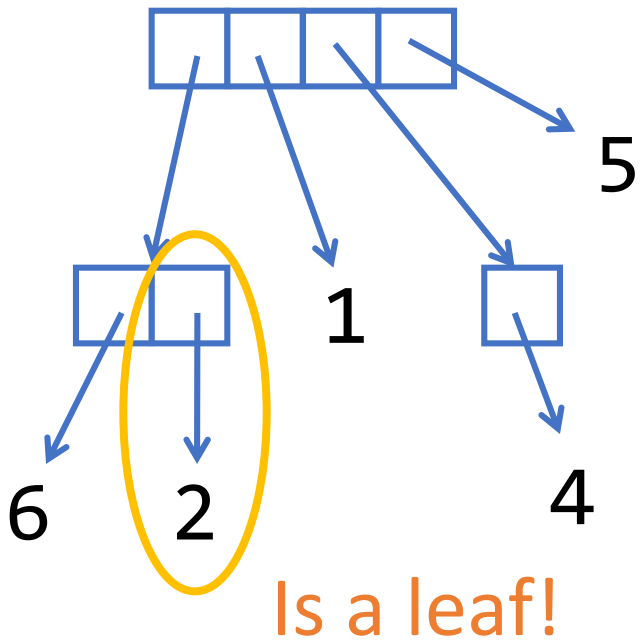
```
tree = ((3, 2), 1, (4,), 5)
```




```
def scale_tree(tree, factor):
    def scale_func(subtree):
        if is_leaf(subtree):
            return factor*subtree
        else:
            return scale_tree(subtree,
                              factor)
    return map(scale_func, tree)
```

Let's see what `scale_tree` does

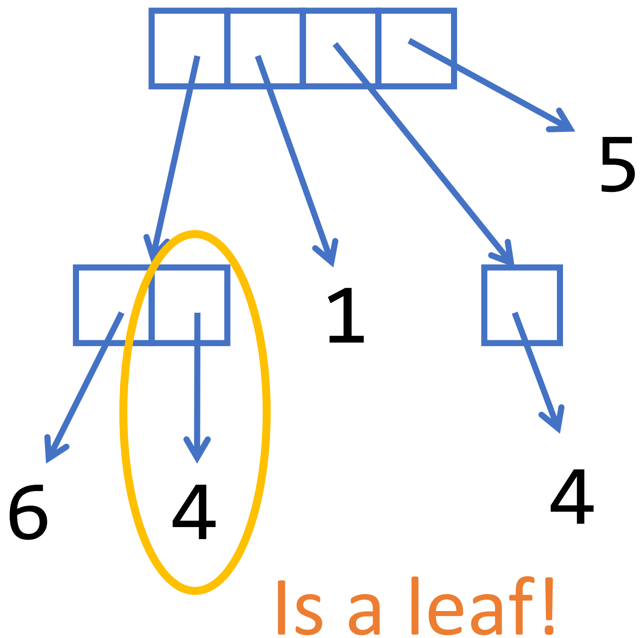
```
tree = ((3, 2), 1, (4,), 5)
```




```
def scale_tree(tree, factor):
    def scale_func(subtree):
        if is_leaf(subtree):
            return factor*subtree
        else:
            return scale_tree(subtree,
                              factor)
    return map(scale_func, tree)
```

Let’s see what scale_tree does

tree = ((3, 2), 1, (4,), 5)

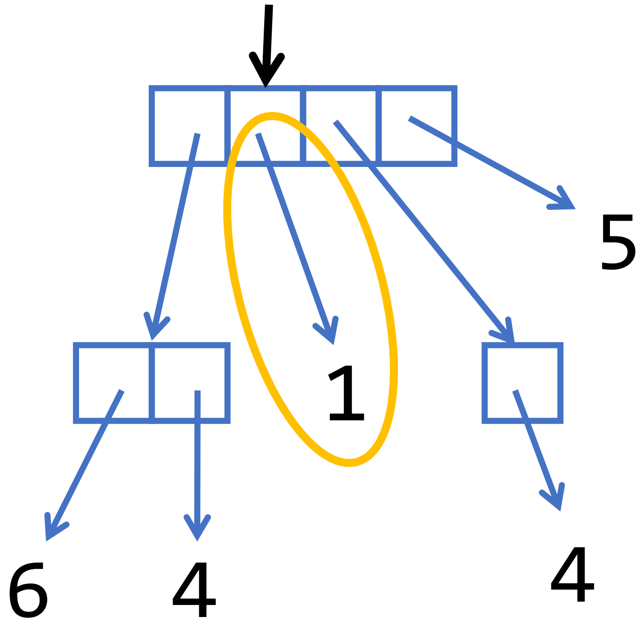



```
def scale_tree(tree, factor):
  def scale_func(subtree):
    if is_leaf(subtree):
       return factor*subtree
    else:
       return scale_tree(subtree,
                         factor)
  return map(scale_func, tree)
```

Let’s see what `scale_tree` does

`tree = ((3, 2), 1, (4,), 5)`

Apply `scale_func` to each element



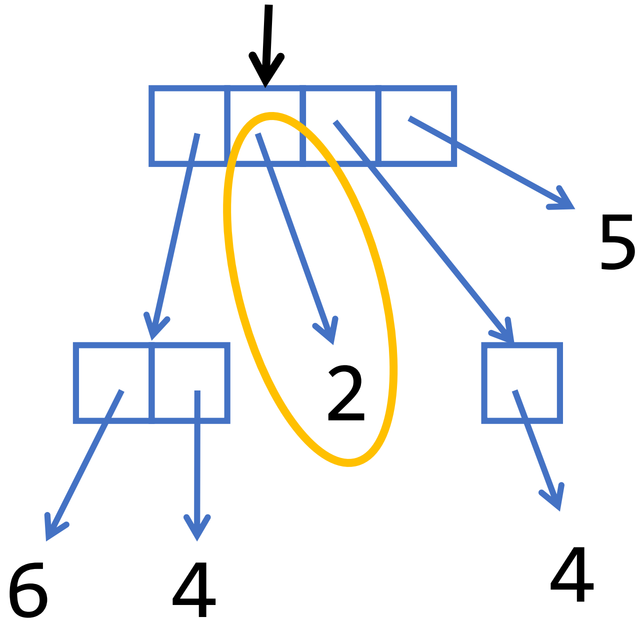
Is a leaf!

```
def scale_tree(tree, factor):
  def scale_func(subtree):
    if is_leaf(subtree):
      return factor*subtree
    else:
      return scale_tree(subtree,
                        factor)
  return map(scale_func, tree)
```

Let’s see what `scale_tree` does

```
tree = ((3, 2), 1, (4,), 5)
```

Apply `scale_func` to each element

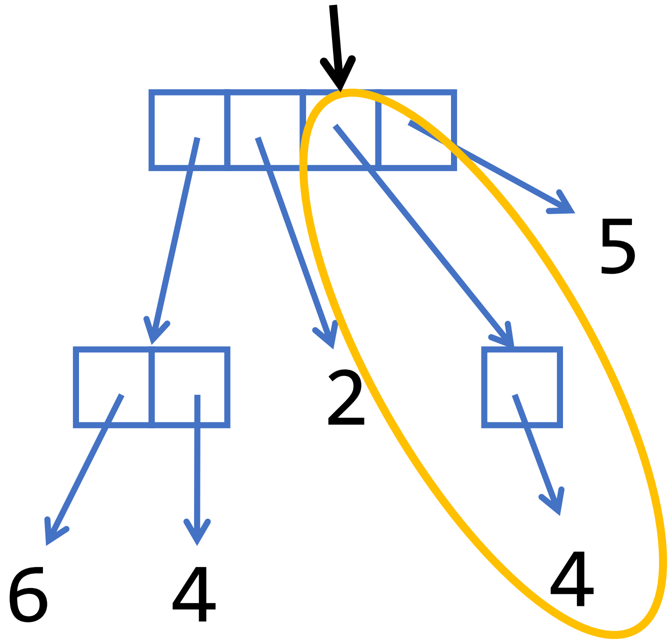



```
def scale_tree(tree, factor):
  def scale_func(subtree):
    if is_leaf(subtree):
       return factor*subtree
    else:
       return scale_tree(subtree,
                         factor)
    return map(scale_func, tree)
```

Let’s see what `scale_tree` does

`tree = ((3, 2), 1, (4,), 5)`

Apply `scale_func` to each element



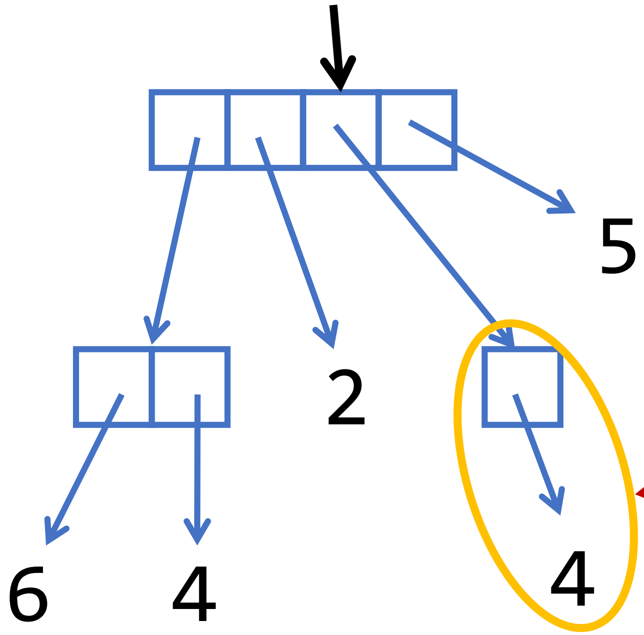
Not a leaf!

```
def scale_tree(tree, factor):
    def scale_func(subtree):
        if is_leaf(subtree):
            return factor*subtree
        else:
            return scale_tree(subtree,
                              factor)
    return map(scale_func, tree)
```

Let's see what `scale_tree` does

```
tree = ((3, 2), 1, (4,), 5)
```

Apply `scale_func` to each element

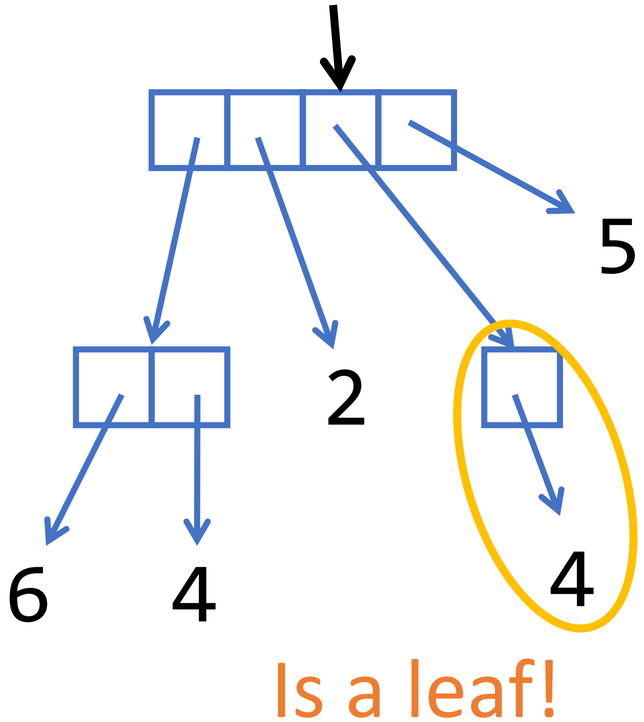



```
def scale_tree(tree, factor):
  def scale_func(subtree):
    if is_leaf(subtree):
      return factor*subtree
    else:
      return scale_tree(subtree,
                        factor)
  return map(scale_func, tree)
```

Let's see what `scale_tree` does

```
tree = ((3, 2), 1, (4,), 5)
```

Apply `scale_func` to each element

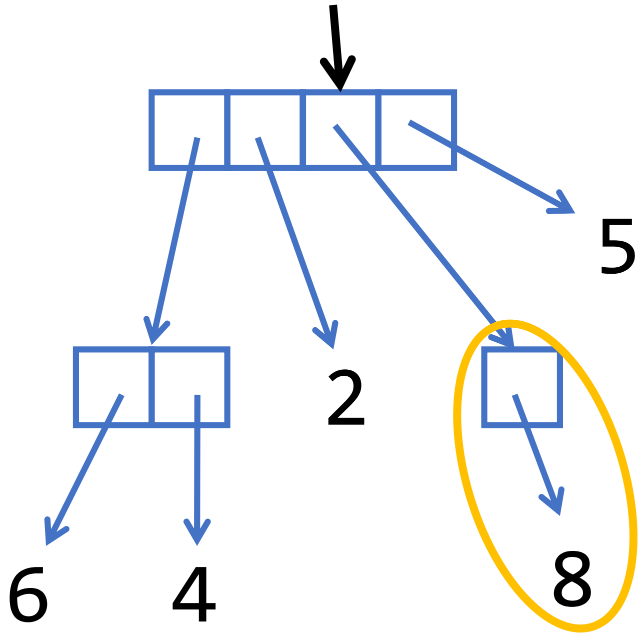



```
def scale_tree(tree, factor):
    def scale_func(subtree):
        if is_leaf(subtree):
            return factor*subtree
        else:
            return scale_tree(subtree,
                              factor)
    return map(scale_func, tree)
```

Let's see what `scale_tree` does

`tree = ((3, 2), 1, (4,), 5)`

Apply `scale_func` to each element

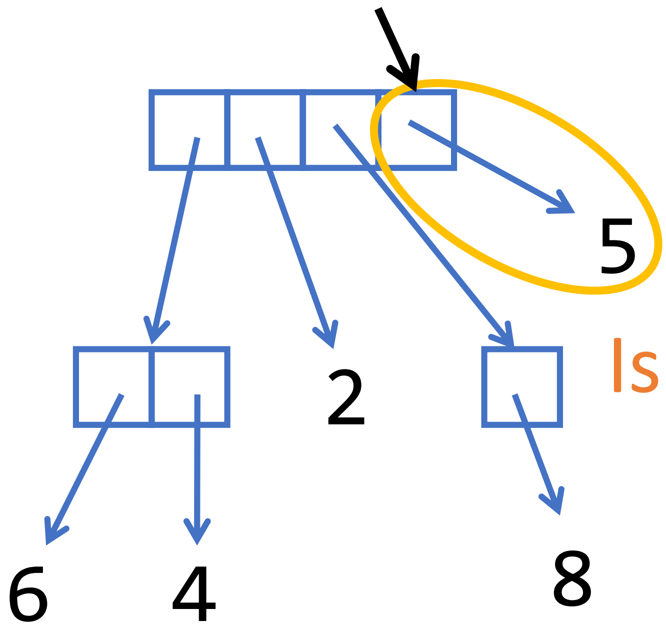



```
def scale_tree(tree, factor):
    def scale_func(subtree):
        if is_leaf(subtree):
            return factor*subtree
        else:
            return scale_tree(subtree,
                              factor)
    return map(scale_func, tree)
```

Let's see what `scale_tree` does

```
tree = ((3, 2), 1, (4,), 5)
```

Apply `scale_func` to each element



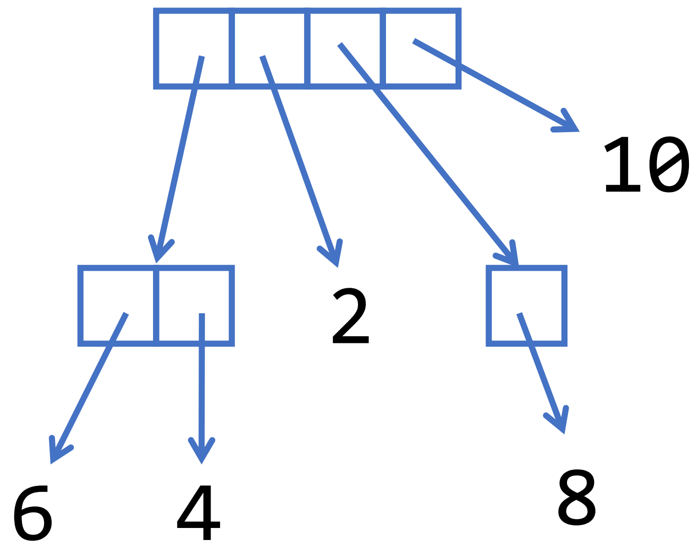
Is a leaf!

```
def scale_tree(tree, factor):
    def scale_func(subtree):
        if is_leaf(subtree):
            return factor*subtree
        else:
            return scale_tree(subtree,
                              factor)
    return map(scale_func, tree)
```

Let's see what `scale_tree` does

```
tree = ((3, 2), 1, (4,), 5)
```

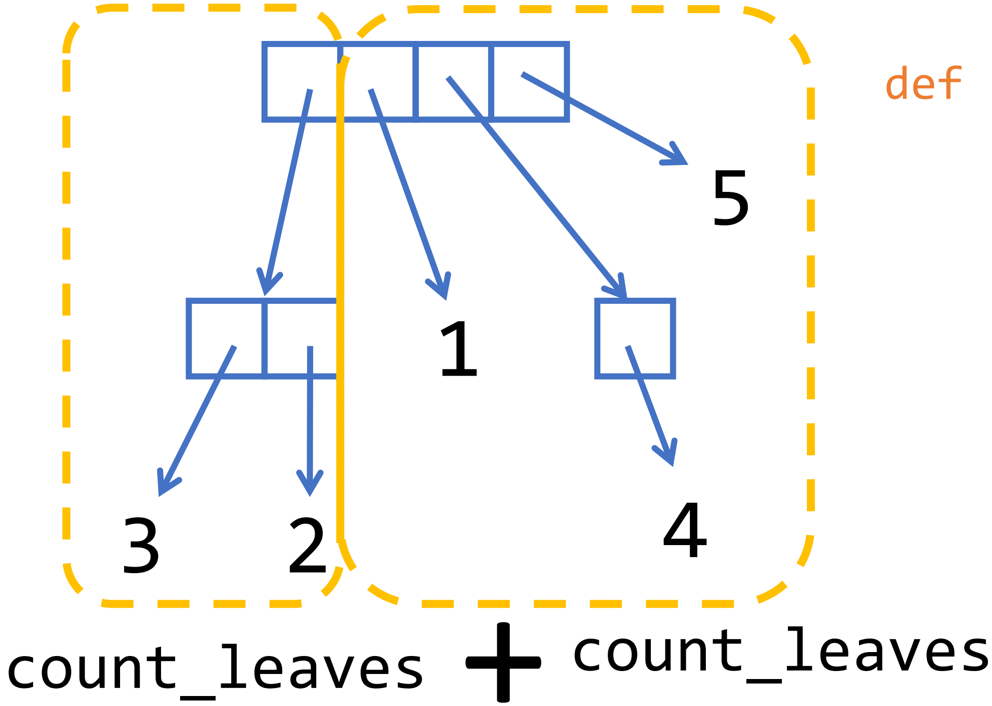
Done applying `scale_func` to each element




```
def scale_tree(tree, factor):
  def scale_func(subtree):
    if is_leaf(subtree):
       return factor*subtree
    else:
       return scale_tree(subtree,
                         factor)
  return map(scale_func, tree)
```

Let's compare with `count_leaves`

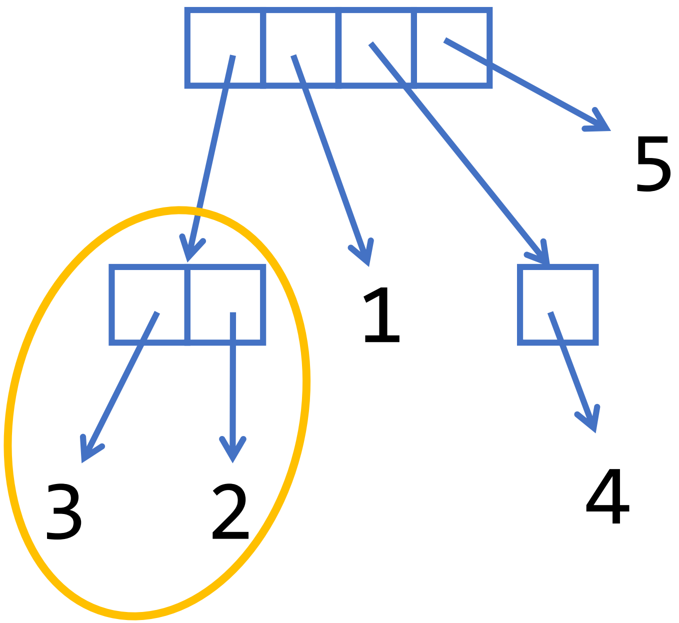
```
tree = ((3, 2), 1, (4,), 5)
```




```
def count_leaves(tree):
    if tree == ():
        return 0
    elif is_leaf(tree):
        return 1
    else:
        return count_leaves(tree[0])
        + count_leaves(tree[1:])
```

Let's compare with `count_leaves`

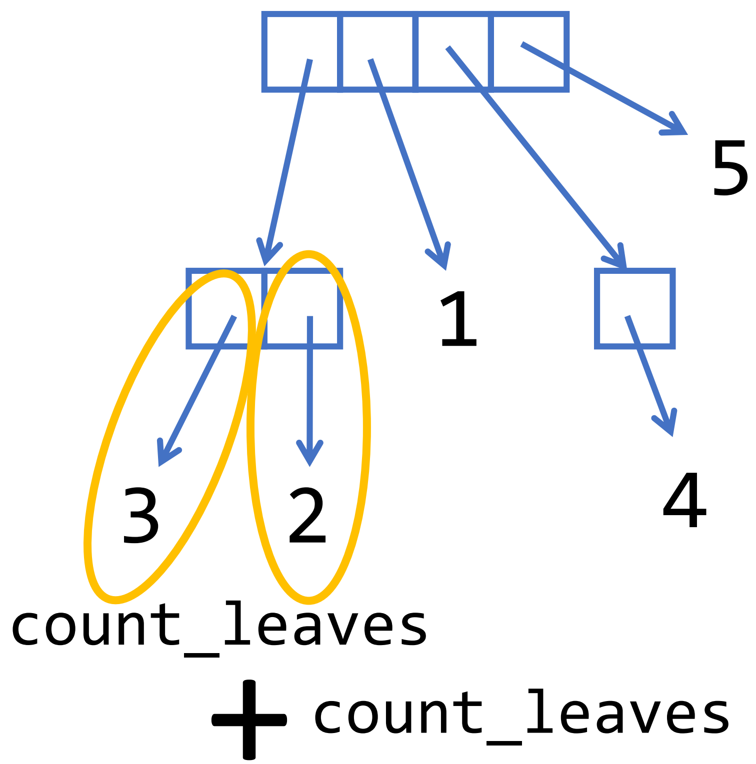
`tree = ((3, 2), 1, (4,), 5)`




```
def count_leaves(tree):
    if tree == ():
        return 0
    elif is_leaf(tree):
        return 1
    else:
        return count_leaves(tree[0])
        + count_leaves(tree[1:])
```

Let's compare with count_leaves

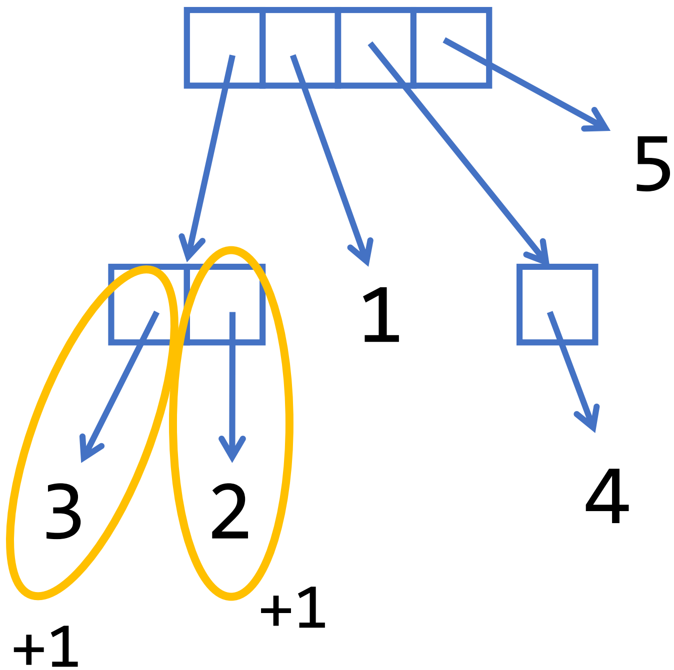
```
tree = ((3, 2), 1, (4,), 5)
```




```
def count_leaves(tree):
    if tree == ():
        return 0
    elif is_leaf(tree):
        return 1
    else:
        return count_leaves(tree[0])
        + count_leaves(tree[1:])
```

Let's compare with `count_leaves`

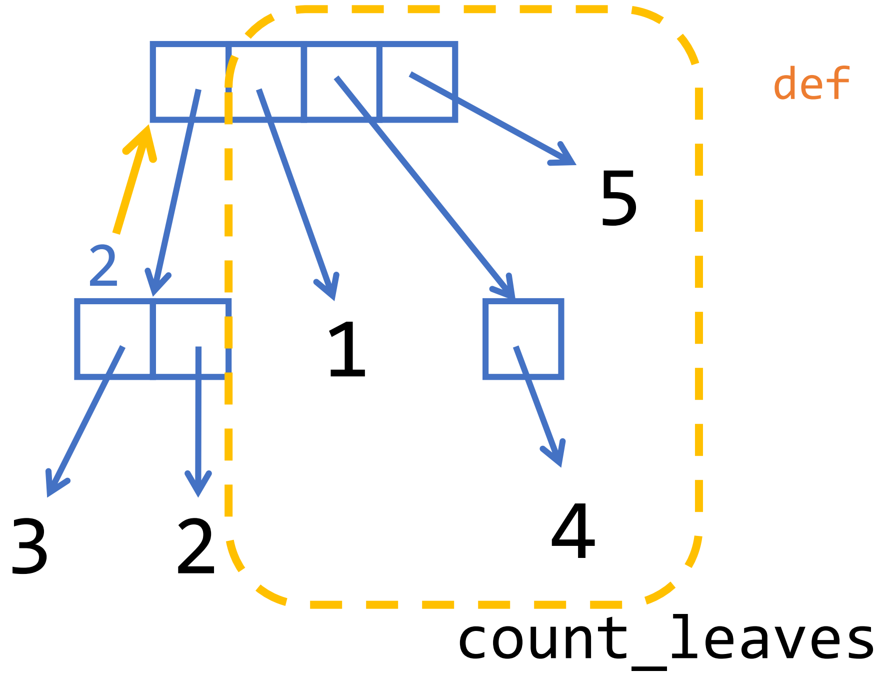
```
tree = ((3, 2), 1, (4,), 5)
```




```
def count_leaves(tree):
    if tree == ():
        return 0
    elif is_leaf(tree):
        return 1
    else:
        return count_leaves(tree[0])
        + count_leaves(tree[1:])
```

Let’s compare with count_leaves

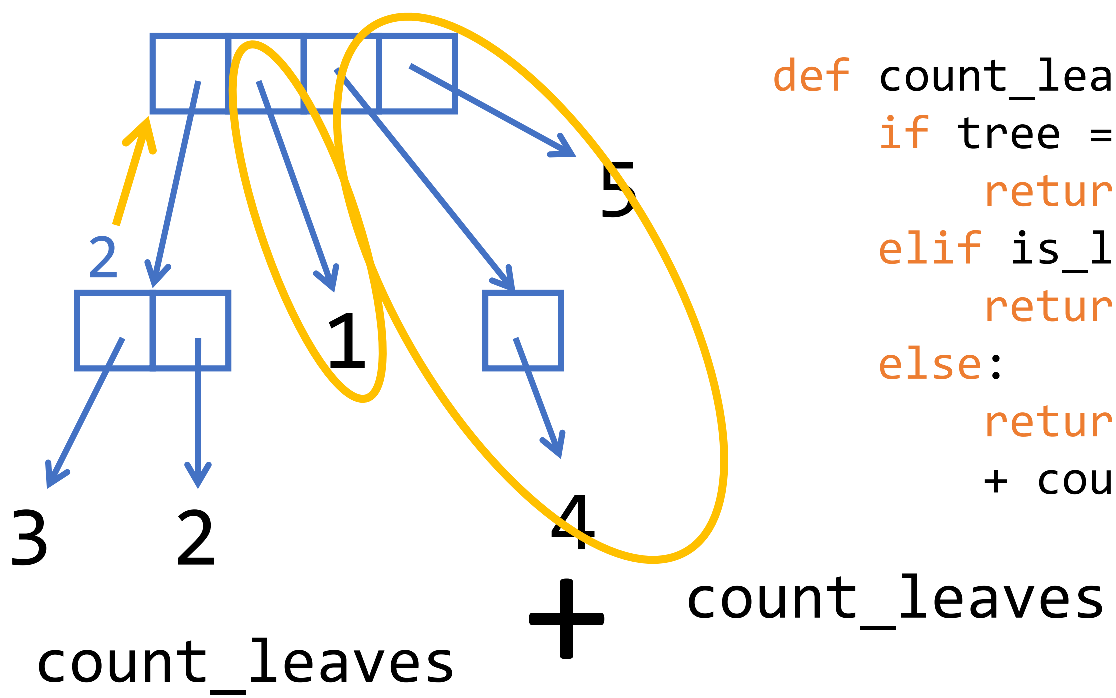
tree = ((3, 2), 1, (4,), 5)




```python
def count_leaves(tree):
    if tree == ():
        return 0
    elif is_leaf(tree):
        return 1
    else:
        return count_leaves(tree[0])
        + count_leaves(tree[1:])
```

Let’s compare with count_leaves

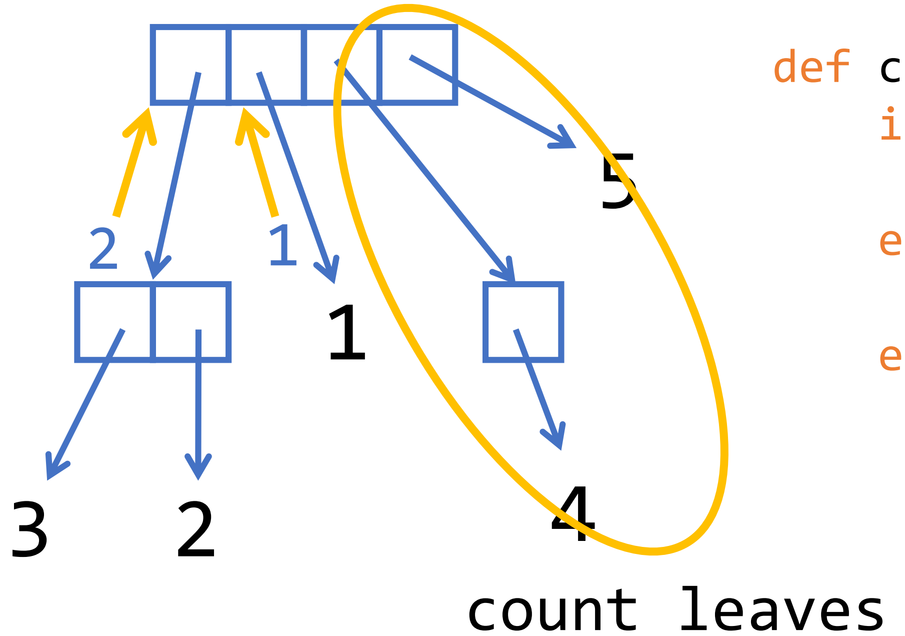
```
tree = ((3, 2), 1, (4,), 5)
```




```
def count_leaves(tree):
    if tree == ():
        return 0
    elif is_leaf(tree):
        return 1
    else:
        return count_leaves(tree[0])
        + count_leaves(tree[1:])
```

Let's compare with `count_leaves`

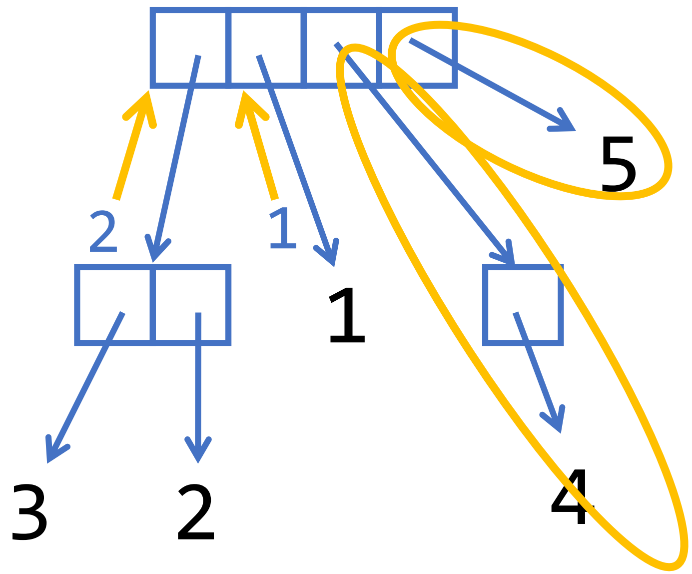
```
tree = ((3, 2), 1, (4,), 5)
```




```python
def count_leaves(tree):
    if tree == ():
        return 0
    elif is_leaf(tree):
        return 1
    else:
        return count_leaves(tree[0])
        + count_leaves(tree[1:])
```

Let’s compare with count_leaves

```
tree = ((3, 2), 1, (4,), 5)
```

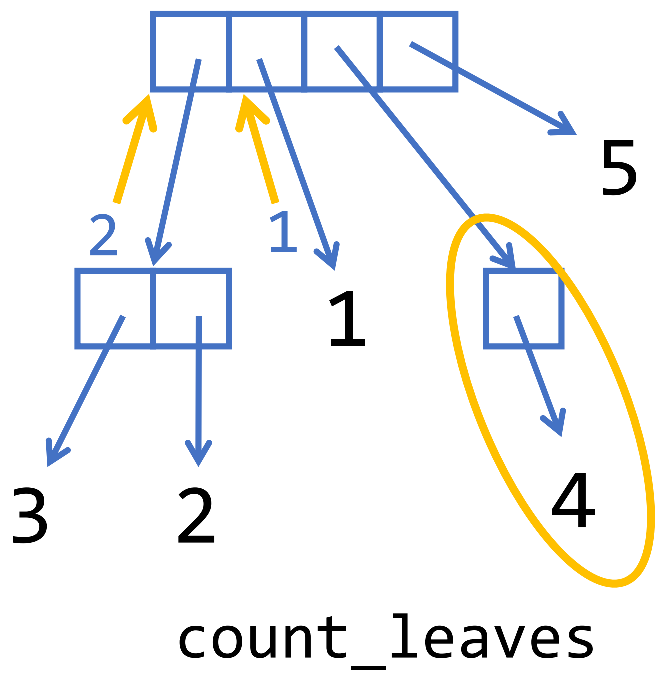


count_leaves + count_leaves

```
def count_leaves(tree):
    if tree == ():
        return 0
    elif is_leaf(tree):
        return 1
    else:
        return count_leaves(tree[0])
        + count_leaves(tree[1:])
```

Let’s compare with count_leaves

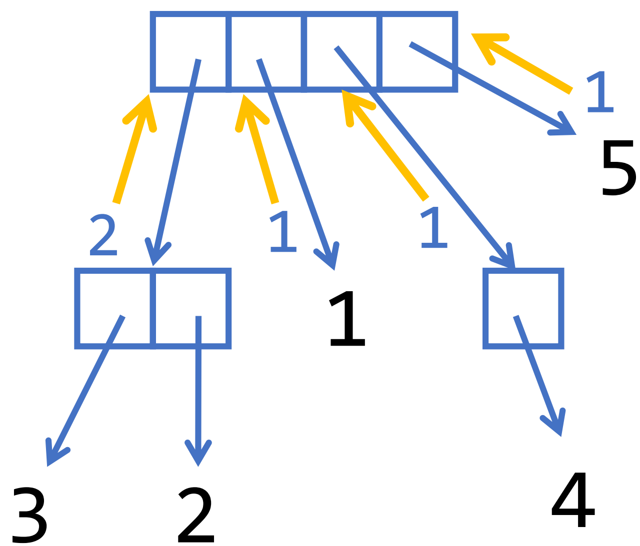
```
tree = ((3, 2), 1, (4,), 5)
```




```
def count_leaves(tree):
    if tree == ():
        return 0
    elif is_leaf(tree):
        return 1
    else:
        return count_leaves(tree[0])
        + count_leaves(tree[1:])
```

Let’s compare with count_leaves

tree = ((3, 2), 1, (4,), 5)




```
def count_leaves(tree):
    if tree == ():
        return 0
    elif is_leaf(tree):
        return 1
    else:
        return count_leaves(tree[0])
        + count_leaves(tree[1:])
```

Key Idea:

Traverse tree with recursion

Check for leaf!

Sanity Check (QOTD)

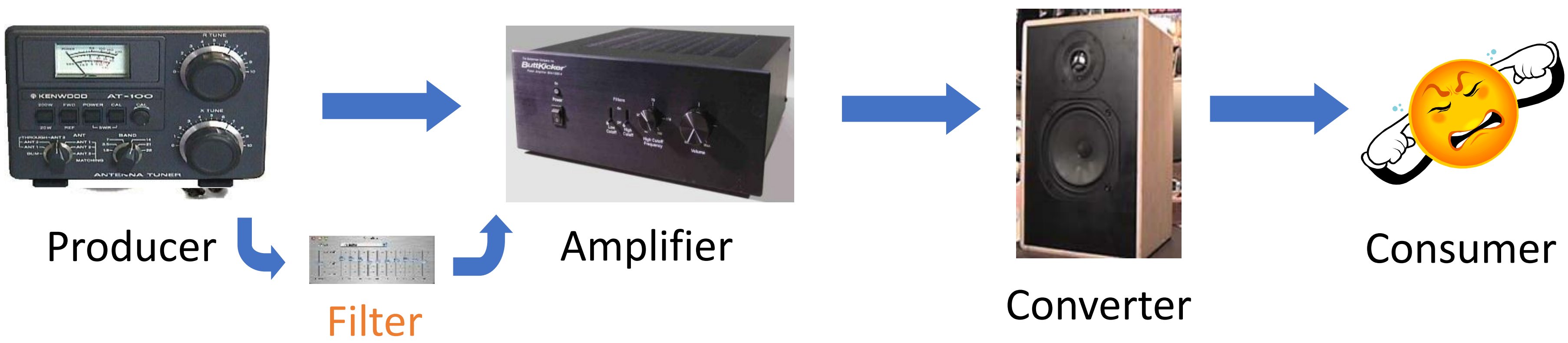
How do you write a function copy_tree that takes a tree and returns a copy of that tree?

Sanity Check (QOTD)

```
def copy_tree(tree):
    return tree # is NOT acceptable!

>>> t = (1, 2, 3)
>>> t_copy = copy_tree(t)
t == t_copy → True
t is t_copy → False
```

Listening to Music

- Signal goes through various stages of "processing".
- Additional component can be inserted.
- Easy to change component.
- Components interface via signals.



Modeling Computation as Signal Processing

- Producer (enumerator) creates signal.
- Filter removes some elements.
- Mapper modifies signal.
- Consumer (accumulator) consumes signal.

Benefits

1. Modularity: each component independent of others; components may be re-used.
2. Clarity: separates data from processes
3. Flexibility: new component can be added

Example:
Sum of squares of odd leaves

Given a tree, want to add the squares of (only) leaves of odd numbers:

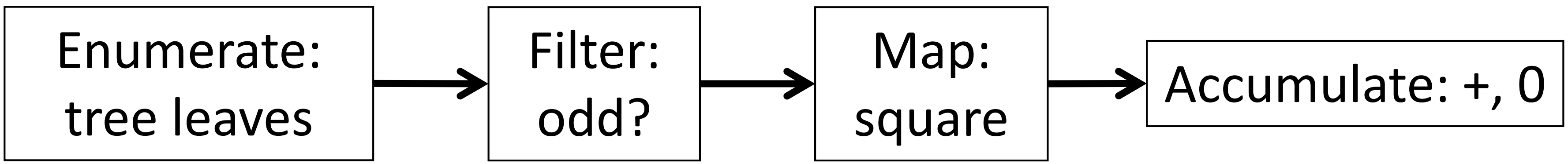
```
sum_odd_squares(((1, 2), (3, 4))) → 10
```

Example:
Sum of squares of odd leaves

```
def sum_odd_squares(tree):
    if tree == ():
        return 0
    elif is_leaf(tree):
        if tree % 2 == 0:
            return 0
        else:
            return tree ** 2
    else:
        return sum_odd_squares(tree[0]) +
               sum_odd_squares(tree[1:])
```

Alternative Approach

View it as signal processing computation!



How to represent "signals"?

- Sequences

Enumerating leaves

What does the following function do?

```
def enumerate_tree(tree):
    if tree == ():
        return ()
    elif is_leaf(tree):
        return (tree,)
    else:
        return enumerate_tree(tree[0]) +
               enumerate_tree(tree[1:])
```

```
enumerate_tree((1, (2, (3, 4)), 5))
 → (1, 2, 3, 4, 5)
```

Also known as flattening the tree.

Filtering a sequence

```
def filter(pred, seq):
    if seq == ():
        return ()
    elif pred(seq[0]):
        return (seq[0],)
               + filter(pred, seq[1:])
    else:
        return filter(pred, seq[1:])
```

Note: we are overwriting the default Python filter function!

```
is_odd = lambda x:x%2 != 0
filter(is_odd, (1, 2, 3, 4, 5))  →  (1, 3, 5)
```

Accumulating a sequence

```
def accumulate(fn, initial, seq):
    if seq == ():
        return initial
    else:
        return fn(seq[0],
                  accumulate(fn, initial,
                             seq[1:]))
add = lambda x, y: x+y
accumulate(add, 0, (1, 2, 3, 4, 5))
accumulate(lambda x, y:(x, y), (),
           (1, 2, 3, 4, 5))
→ (1, (2, (3, (4, (5, ()))))) → 15
```

Putting it together

```python
def sum_odd_squares(tree):
  accumulate(add, 0,
             map(square,
                 filter(is_odd,
                        enumerate_tree(tree))))
```

Enumerate: tree leaves → Filter: is_odd → Map: square → Accumulate: add, 0

Putting it together

```
(1, 2, (3, 4))
```

enumerate_leaves ↓

```
(1, 2, 3, 4)
```

filter odd? ↓

```
(1, 3)
```

map square ↓

```
(1, 9)
```

accumulate add , 0 ↓

```
10
```

Another Example: Tuple of even Fib

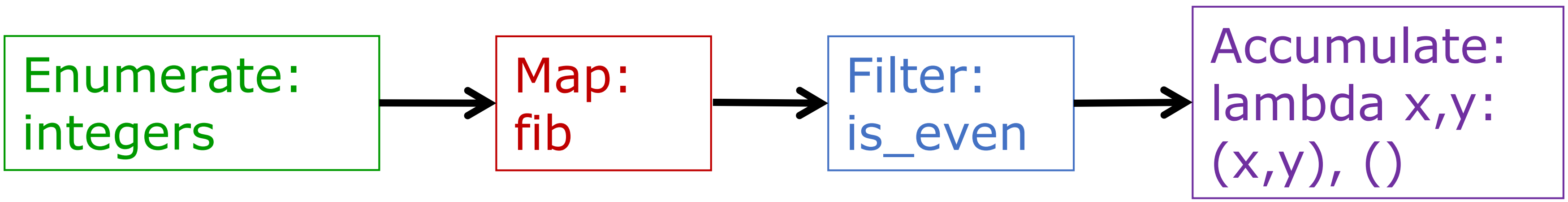
Want a list of even $fib(k)$ for all k up to given integer n.

“Usual” Way

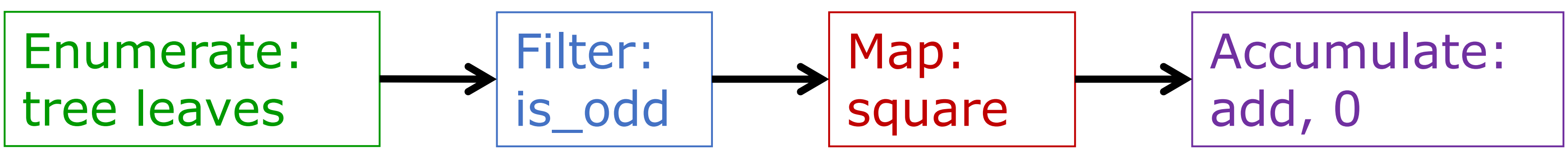
```
def even_fibs(n):
    result = ()
    for k in range(1, n + 1):
        f = fib(k)
        if is_even(f):
            result = result + (f, )
    return result

is_even = lambda x: x % 2 == 0

>>> even_fibs(30)
(2, 8, 34, 144, 610, 2584, 10946, 46368, 196418,
832040)
```

Signal processing view

- Even fibs



- Compare: sum square odd leaves

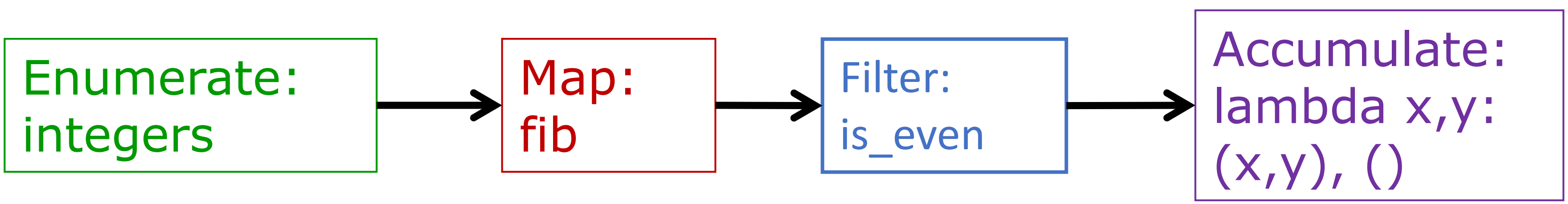


Enumerate integers

```
def enumerate_interval(low, high):
    return tuple(range(low,high+1))

enumerate_interval(2, 7)
→ (2, 3, 4, 5, 6, 7)
```

Even Fibs

```
def even_fibs(n):
    accumulate(lambda x,y: (x,y), (),
          filter(is_even,
            map(fib,
               enumerate_interval(1, n))))
```



Signal Processing View

- Modular components:
  - Enumerate, Filter, Map, Accumulate
  - Each is independent of others.
  - Modularity is a powerful strategy for controlling complexity.

Signal Processing View

- Build a library of components.
- Sequences used to interface between components.

Re-using components

- Want a sequence of squares of fib(k)

```
def list_fib_squares(n):
  return accumulate(lambda x,y: (x, y), (),
                    map(square,
                        map(fib,
                            enumerate_interval(1, n))))
```

Other Uses

- Suppose we have a sequence of personnel records.
- Want to find salary of highest-paid programmer.

```
def salary_of_highest_paid_programmer(records):
  return accumulate(max, 0,
                    map(salary,
                        filter(is_programmer, records)))
```

Default Python map and filter functions

Returns an iterable instead of tuple, but you can force it into a tuple.

```
>>> a = (1,2,3,4,5)
>>> b = filter(lambda x: x%2 == 0, a)
>>> b
<filter object at 0x02EC4710>
```

```
>>> for i in b:
        print(i)
2
4

>>> c = tuple(b)
>>> c
()
```

```
>>> b = filter(lambda x: x%2 == 0, a)
>>> b
<filter object at 0x02E42C10>

>>> c = tuple(b)
>>> c
(2, 4)

>>> for i in c:
        print(i)
2
4
```

Working with Files

Reading a File:

```
input = open('inputfilename.txt', 'r’)
some_line = input.readline()
```

We can check for end of file by checking whether

```
some_line == '' #empty string
```

Writing to a File:

```
output = open('outputfilename.txt', 'w')
output.write('HELLO WORLD')
```

Example

```python
def metrics(dictfile):
    dict = open(dictfile, 'r')

    currword = dict.readline()
    longest_word = currword
    shortest_word = currword

    while currword != '':
        if(len(currword) < len(shortest_word)):
            shortest_word = currword
        if(len(currword) > len(longest_word)):
            longest_word = currword
        currword = dict.readline()

    output = open("output.txt", "w")
    output.write("longest word: " + longest_word)
    output.write("shortest word: " + shortest_word)
```

Find longest and shortest word

write to file

Example

```
dictionary.txt >>
CS1010S
BEST
MODULE
WORLD

metrics("dictionary.txt")

output.txt >>
longest word: CS1010S
shortest word: BEST
```

Summary

- Data often comes in the form of sequences
  - Easy to manipulate using recursion/iteration
  - Can be nested
- Closure property allows us to build hierarchical structures, e.g. trees, with tuples
  - Can use recursion to traverse such structures

Summary

- “Signal-processing” view of computation.
  - Powerful way to organize computation.
  - Sequences as interfaces
  - Components: (i) Enumeration, (ii) Map, (iii) Filter, (iv) Accumulate

WebEx Test Session